CouchDB

Overview

Giuseppe Prisco 1895709

MATERIAL FOR THE PRESENTATION

The contents of the presentation had been taken and reworked from the following sources:

- **CouchDB Official Documentation**: https://docs.couchdb.org/en/stable/
- **Apache CouchDB Wiki**: https://cwiki.apache.org/confluence/display/couchdb/introduction

STRUCTURE OF THE PRESENTATION

1. INTRODUCTION
   - Description
   - CouchDB and HTTP
2. DOCUMENTS
   - Structure
   - Document version
   - Attachments
3. CONSISTENCY, ACID PROPERTIES
   - Local and Distributed Consistency
   - CouchDB and ACID
4. VIEWS
   - View Model
   - Design Document
   - MapReduce & other functions
5. REPLICATION
   - Replication Types
   - Replication Document
   - Conflicts
6. SHARDING, PARTITIONS
   - Clusters
   - Shard Management
   - Partitioned Databases

1

INTRODUCTION

Description,
CouchDB and HTTP

Introduction to CouchDB

- **CouchDB** is an open source NoSQL document database which stores data in **JSON documents**
- Each CouchDB document has its own implicit structure (**schema-free** data model)
- CouchDB provides a **RESTful HTTP API** for reading, updating and deleting database documents
- Some ways to communicate with CouchDB are the **command-line utility curl** or the built-in **web interface Fauxton** (more on them later)

2

DOCUMENTS

Structure,
Document version,
Attachments

DOCUMENT STORAGE

- Each database in CouchDB stores data as **JSON documents**
- Documents consist of any number of **uniquely named fields** containing values of various types (number, string, boolean, array ecc.)
- Every document is uniquely identified in the database with a **special field** named "**_id**"
- This id represents a **Universally Unique Identifier** (UUID) and can be set both manually and automatically by CouchDB

DOCUMENT VERSION

- Another field automatically set by CouchDB is “**_rev**”
- In order to **modify a value** of a specific field, one has to save the entire new version (**revision**) of that document into CouchDB
- When updating a document, we need to include the “_rev” field of the version we want to change
- The revision value consists of the **MD5 hash of the document** with an integer prefix representing the number of times the document has been updated

DOCUMENT VERSION (cont.)

- This mechanism allows CouchDB to not accept changes made on documents that were modified before submitting the update
- This revision system is called **Multi-Version Concurrency control** (MVCC) and is also used for **replication purposes**
- However the important **difference with a version control system** is that CouchDB does not guarantee that older versions of a document are kept

DOCUMENTS - ATTACHMENTS

- An additional field that a document can have is “**_attachments**”
- Attachments are identified by their **name**, their **MIME/Content type** and the **number of bytes they contain**
- Attachments can be of **any data**, including images, text, videos, music files ecc.
- Every attachment is characterized by their **own URL** and can be accessed as any other web resource

Example of a CouchDB document

```
{
    "_id": "6e1295ed6c29495e54cc05947f18c8af",
    "_rev": "3-131533518",
    "title": "There is Nothing Left to Lose",
    "artist": "Foo Fighters",
    "year": "1997",
    "_attachments": {
        "artwork.jpg": {
            "stub": true,
            "content_type": "image/jpg",
            "length": 52450
        }
    }
}
```

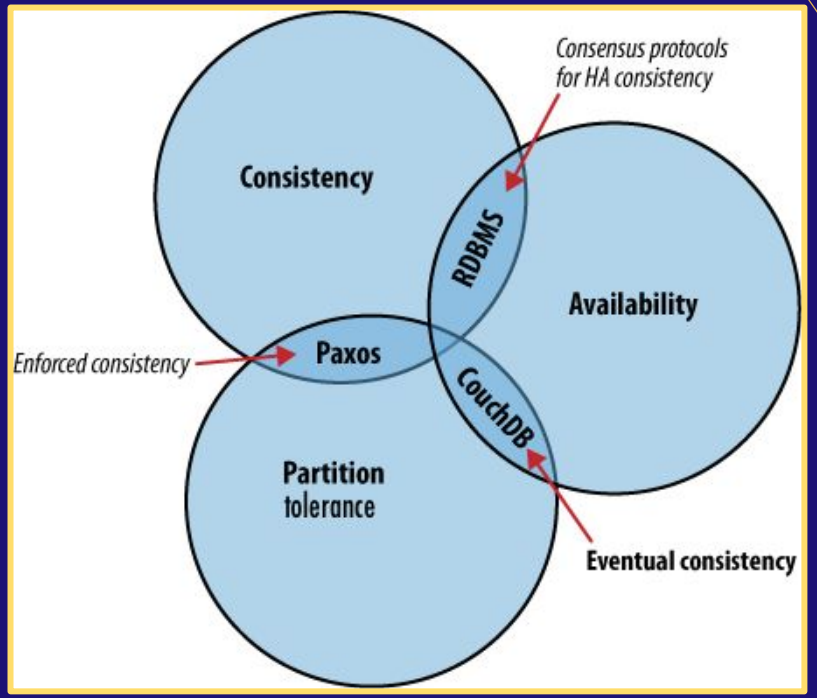
3

CONSISTENCY, ACID PROPERTIES

Local and Distributed
Consistency,
CouchDB and Acid

CONSISTENCY

- CouchDB enforces “**Eventual Consistency**” instead of immediate consistency
- As a result, the system is **highly available**



CONSISTENCY (cont.)

Depending on the particular scenario, consistency is achieved through different concurrency mechanisms.
We distinguish between **single nodes** and **clusters**

Local Consistency

In a single node, CouchDB uses **MVCC** to manage concurrent access to a database

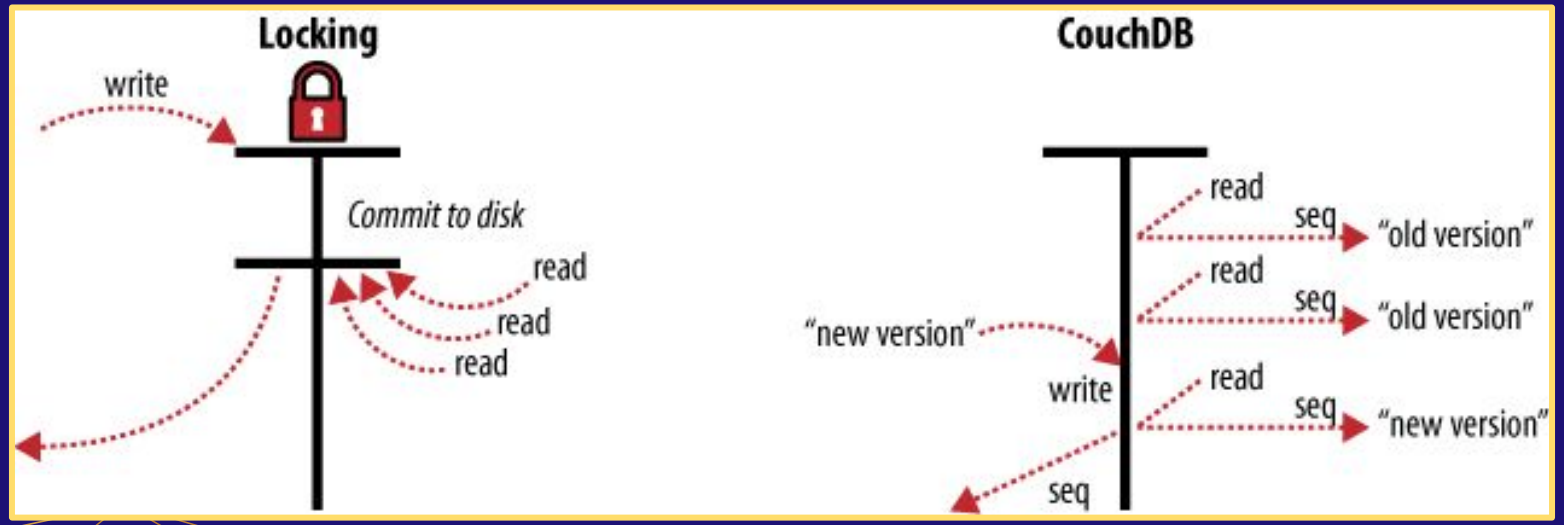
No lock system is used (as in RDBMS)

Distributed Consistency

In a cluster, consistency is achieved by using **incremental replication** (see later)

This process allows document changes to **periodically** be copied between servers

CONSISTENCY (cont.)



ACID PROPERTIES

- Document updates are serialized and **clients are never locked out** of reading a document, even by concurrent updates
- Read operations use MVCC model in which each client sees a **consistent snapshot of the database** for the entire duration of the read
- CouchDB can ensure transactional and ACID properties on a per-document basis

4

VIEWS

View Model,
Design Documents,
MapReduce & other functions

VIEW MODEL

- In order to **organize**, **filter** and **report** on data, CouchDB integrates a **view model**
- Views **are made on-demand** to **join** and **aggregate** on database documents
- Since views don’t affect the documents and are built dynamically, we can have **many different view** representations for the **same set of data**

VIEW MODEL (cont.)

- Views are separate entities from the data they display and are defined inside a special type of documents: **design documents**
- Views are defined with JavaScript functions that employ the **map-reduce** system
- Each view function **takes as input a document** and does computations to display data that needs to be made available through the view
- The **view engine maintains indexes of its views** and incrementally updates them as changes in the database are made

DESIGN DOCUMENTS

Design documents contain functions such as **view**, **update** and **filter** functions. An example of their structure is the following:

```
{
    "_id": "_design/example",
    "views": {
        "view-number-one": {
            "map": "function (doc) {/* function code here */}"
        },
        "view-number-two": {
            "map": "function (doc) {/* function code here */}",
            "reduce": "function (keys, values, rereduce) {/* function code here */}"
        }
    },
    "updates": {
        "updatefun1": "function(doc,req) {/* function code here */}",
        "updatefun2": "function(doc,req) {/* function code here */}"
    },
    "filters": {
        "filterfunction1": "function(doc, req){ /* function code here */ }"
    },
    "validate_doc_update": "function(newDoc, oldDoc, userCtx, secObj) { /* function code here */ }",
    "language": "javascript"
}
```

VIEW FUNCTIONS - MAP REDUCE

- To compute the result of a view CouchDB uses **MapReduce**
- MapReduce utilizes two functions, “**map**” and “**reduce**”, that are **applied to each document** in isolation
- The ability to isolate these operations render the computation of the view easily **parallelizable**
- Additionally, since the result of MapReduce produces key-value pairs, **CouchDB uses a B-tree index** (sorted by key) to make lookup or range operations much more efficient

VIEW FUNCTIONS - MAP

The **map** function takes a document as the argument and **emits key-value pairs** that are stored in the view

```
function (doc) {
  if (doc.type === 'post' && doc.tags && Array.isArray(doc.tags)) {
    doc.tags.forEach(function (tag) {
      emit(tag.toLowerCase(), 1);
    });
  }
}
```

- To prevent one map function to change the document's state and another one receiving a modified version of it, each **document is sealed**
- For efficiency purposes, each document is processed by a group of map functions from all the views of a design document meaning that an index update for one view also triggers the update for the others

VIEW FUNCTIONS - REDUCE

The **reduce** function takes two required arguments (keys and values of the related map function) and **reduces the mapped result**

```
function (keys, values, rereduce) {
  return sum(values);
}
```

- There exists also the Rereduce mode, used for additional reduce values list
- Additionally there is the possibility to have results longer than the initial values list (however it is not suggested outside debug purposes) by setting the “reduce_limit” config option to “false”

BUILT-IN REDUCE FUNCTIONS

CouchDB has also a set of **built-in reduce functions**:

- **_approx_count_distinct**: approximates the number of distinct keys in a view index
- **_count**: counts the number of values with a given key in the index
- **_stats**: computes some quantities for numeric values associated with a key, including sum, min, max and count
- **_sum**: sums the numeric values associated with each key

UPDATE AND FILTER FUNCTIONS

Design documents also contain **update functions** and **filter functions**

Update Functions

These are functions that the client can request to **create or update** a document **server-side**

If the request contains the ID of a document in the URL, the server will provide the newest version of it

Filter Functions

They return “true” if the document **passes the rules of the filter**

There exist **Classic Filters** (also dynamic) and **View Filters** (which use the map function instead of the filter function)

VALIDATION FUNCTIONS

- A design document can contain a function called “**validate_doc_update”**
- CouchDB uses this function to **prevent invalid** or **unauthorized** document **updates** from proceeding
- Typically, validation functions analyze the structure of the document to **check that required fields are present** and that the **user is allowed to make changes**
- The validation function (which is optional) **is executed for each document to be created or updated**. If it raises an exception the update is denied, otherwise it will get accepted

5

REPLICATION

Replication Types,
Replication Document,
Conflicts

REPLICATION

- The **replication process** involves a **source database** and a **destination database**
- At the end of this process all **documents** present **in the source** database **will** also **be on the destination** database. Moreover all the deleted documents of the source are also deleted in the destination
- Replication involves only **the last revision of a document** (any previous revision is ignored)

REPLICATION TYPES

There exist **two methods** to enable replication

Transient Replication

In this kind of replication there aren't any documents backing up the replication

This means that upon a **server's restart** the **replication will disappear**

Persistent Replication

In the case of persistent replication, there exists a "**_replicator**" database

It is used to **keep replication parameters** after the server restarts

REPLICATION PROCEDURE

- CouchDB first **compares the two databases** to check which documents are different using the “Changes Feeds”
- Once it reaches the end of the changes feed, the replication task finishes
- There is the possibility to **make the task continue to wait** for new changes by setting the “**continuous**” property to “true”
- In order to achieve a **Master-Master** replication we can set **replication tasks in both directions**

REPLICATION RULES

There exist **three possibilities** to control **which documents are replicated**:

1. **Local documents**: by defining these documents, they will not be replicated

2. **Selector Objects**: these objects can be included in a replication document and contain a query expression to check if a document needs to be replicated

3. **Filter Functions**: the replication task will evaluate the filter function on the documents in the changes feeds and only those that pass the filter check will be replicated

REPLICATION DOCUMENT

A replication document is located into the “**_replicator**” **database**.
An example of its structure is the following:

```
{
    "_id": "my_rep",
    "source": "http://myserver.com/foo",
    "target": {
        "url": "http://localhost:5984/bar",
        "auth": {
            "basic": {
                "username": "user",
                "password": "pass"
            }
        }
    },
    "create_target":  true,
    "continuous": true
}
```

CONFLICTS

- When replicating documents, many “**conflicts**” may arise
- CouchDB allows for any number of conflicting documents to exist within the database but only one of them will be considered the “winner” while the others are tagged as conflicts
- However, we can still access the set of conflicts and they can be replicated as normal, leaving **conflict resolution to the application**
- CouchDB maintains a list of **previous revisions** of updated documents into a structure named “**Revision Tree**” (since when conflicts are introduced, the history branches into a tree)

6

SHARDING, PARTITIONS

Clusters,
Shard Management,
Partitioned Databases

CLUSTERS

In the “etc/default.ini” file in CouchDB there is a section named “**cluster**”:

```
[cluster]
q=2
n=3
```

- “**q**” is the **number of shards**
- “**n**” represents the **number of replicas** for each document, that is the number of copies of the document

For a CouchDB cluster with **default values**, there can be at most **6 nodes** (q*n; 2*3=6) that **can be used to scale out**

NODE/DATABASE MANAGEMENT

To **view the node's name** and all other nodes it is connected to we can use:

```
curl -X GET "http://xxx.xxx.xxx.xxx:5984/_membership" --user admin-user
```

To **add a node** we can use the following (to remove it the command is analogous with the difference that we also include the revision):

```
curl -X PUT "http://xxx.xxx.xxx.xxx/_node/_local/_nodes/node2@yyy.yyy.yyy.yyy" -d {}
```

To **create a database** we can use the following:

```
curl -X PUT "http://xxx.xxx.xxx.xxx:5984/database-name?n=3&q=8" --user admin-user
```

SHARD MANAGEMENT

- Depending on the cluster's size or the number of shards and replicas, **a node may not have access to every shard**
- However **every node** can use **CouchDB' internal shard map** to be able to know where **all the shards' replicas are located**
- When a **request** comes to a CouchDB cluster, it is processed by a **random coordinating node**
- The coordinating node proxies the request to other nodes and once a **quorum** of database nodes have responded, it replies to the client
- The **default size** for the quorum is **r=w=((n+1)/2)** where **r** is the **read quorum**, **w** is the **write quorum** and **n** is the **number or replicas** of each shard

PARTITIONED DATABASES

- In a **partitioned database** documents are assigned to a partition and are given a **partition key**
- The main benefit of using a partitioned database is to **access secondary indices much more efficiently** when locating matching documents since they are all contained inside a single partition
- This results in a single partition range scan when using an index for a read operation instead of reading from a copy of every shard
- It is logical to **group documents** by a **field of interest** (for example the location) and this field will be the **partition key**

PARTITIONED DATABASES (cont.)

- By placing an entire group of documents within a specific shard, the **view engine will have to consult only one copy of the shard** when executing a query
- Thus, **not all the** “**q**” number of **shards** present in the database **will be queried**
- Furthermore, we **do not have to wait for all** “**q**” **shards** to issue a response
- The default value for the maximum size for a partition is 10 GB and can be modified with the “max_partition_size” option

THE END